COMS4115 Fall 2011 Project Proposal

Setup: A Language for Operating on Sets

Adam Weiss	Andrew Ingraham	Ian Erb	Bill Warner
ajw2137@columbia.edu	aci2110@columbia.edu	ire2102@columbia.edu	whw2108@columbia.edu

1. Introduction and Motivation

Setup defines a syntax for operating on finite sets. *Setup* provides intuitive notation for quickly and clearly defining sets, as well as performing rudimentary set operations on user-defined sets. *Setup* also defines a notation for for functions which take literals and sets as parameters.

Setup provides a level of abstraction to the user which makes set manipulation more intuitive. We anticipate users will solve simple set-oriented problems like schedule, rudimentary databases, and probability problems.

2. Language Features

2.1 Data Types

Literals / Atoms

- Integers -- [0-9]+
- Float -- Integer.[Integer]+ uses the 32 bit IEEE range
- Character -- A - z, no punctuation or white space
- Strings -- [Character]+
- Symbols -- Globally unique names that may be members of Sets or Tuples

Sets : homogeneous, all elements of the same type, unique values
Tuples or Lists : ordered lists, heterogeneous, can be of mixed type, duplicate values permitted

2.2 Keywords and Operators

Setup allows for the usual four operations {+, - , *, / } on integer and float types, as well as the following operators for set types:

Setup Operator	Mathematical Symbol	Description
intersect	∩	computes the intersection of the sets to the left (lhs) and right (rhs) of it
union	∪	computes the union of lhs and rhs
minus	-	returns lhs with any members of rhs removed
cross	×	returns the cartesian product of lhs and rhs
in	∈	iterates over members of rhs
not	~	returns complement
#	|S|	returns number of elements in S as an int
:=	≔	assignment

sum	□	operates only on numeric sets and returns sum of elements (done coordinate-wise) in the set
and		arranges cross product pairings from sets on the left and right
...		range operator applies to integers and characters
{}		denotes a set of elements
()		denotes an ordered list, or tuple. the cross product of two sets is a set of tuples.
|		where, as in SQL. in a *Setup* clause, the expression to the left of of | declares variable names and their structural relationships, while the expression on the right binds variables to values
--		begins a comment. comments begin with -- and end with a new line
.		converts lhs and rhs to string representation and returns their concatenation. (all types have a string representation)
*****		wildcard is a placeholder that accepts any value without binding it to a variable name or checking its type
;		statement terminator
[]		in function declarations, groups input arguments and statements in function body

3. Functions

We anticipate functions having no side effects on their arguments. Functions accept as arguments literals and their containers (i.e., sets. sets of sets).

3.1 Function Syntax

3.1.1 Definition

```
function FuncName [set x, int c] returns set
[
    statement;
    statement;
    return ret;
]
```

3.1.2 Invocation

```
FuncName [Week, 7];
```

4. Sample Code

4.1 Set Initialization

4.1.1 Initialization using literals and tokens:

```
Hours := { 1 ... 24 };
Weekdays := {Mo Tu We Th Fr};
Weekend := {Sat Sun};
```

4.1.2 Initialization Built-in Operators:

```
FullWeek := Weekdays union Weekend;
    -- {Mo Tu We Th Fr Sat Sun}

WeekdayHrs := Weekdays cross Hours;
    -- {(Mo 1) (Mo 2) ... (Fr 24)}
```

4.1.3 Initialization Using Relations:

```
WeekdayHrs := {(x y) | x in Weekdays and y in Hours};
    -- {(Mo 1) (Mo 2) ... (Fr 24)}

 TokenWeekdayHrs := {"day". str(x) . "-hr" . str(y) | x in Weekdays and y in
Hours};
    -- {dayMo-hr1 ... dayFr-hr24}

MondayHrs := (Mo *) in WeekdayHrs;
    -- {(Mo 1) (Mo 2) ... (Mo 24)}

Hours := {x | (* x) in WeekdayHrs};
    -- {1 ... 24}

TreeWeek := { (d {h}) | d in Weekdays and h in Hours }
    -- {(Mo {1 ... 24}) ... (Fr {1 ... 24})}
```

4.2 Sample Program

Users may want to use *Setup* to solve problems related to probability. The following program computes the expected value of a roll of a fair dice. It can be extended simply to solve harder problems relating to conditional probability and random walks.

Program

```
function ExpVal [ set S ]
[
    Temp := {x*y | (x y) in S};
    return sum Temp;
]

Pips := {1 2 3 4 5 6};
Prob := { 1/6 };

Dice := Pips cross Prob;     -- {(1 1/6) ... (6 1/6)}
print ExpVal [ Dice ];       -- 3.5
```

Output

```
3.5
```